WEB TECHNOLOGIES
Express

Prof. Spurthi N Anjan
Department of Computer Science and Engineering
spurthianjan@pes.edu

An Introduction to Web Services

A Web service is a software system that supports ***interoperable*** machine-to-machine ***interaction***.

- ➢ Interaction means that more than one application is involved.
- ➢ Interoperable means that applications can operate with one another without sharing the same platform, operating system, programming language, etc.

Web services are largely delivered by a set of protocols: SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language), and UDDI (Universal Description, Discovery, and Interoperability). These three build on XML (the metalanguage for the representation) and HTTP, (the transport protocol).

Components of Web Services

SOAP (Simple Object Access Protocol):

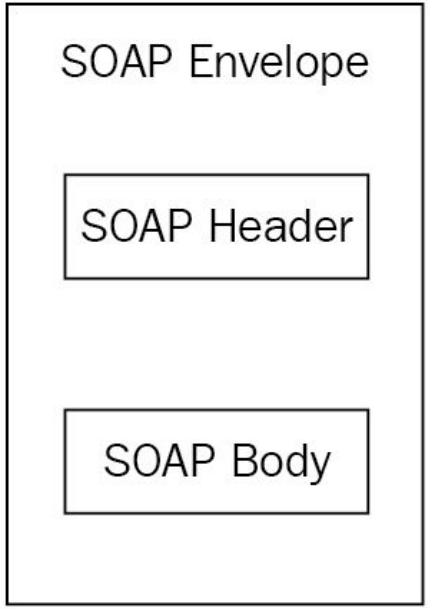
SOAP defines a uniform way of passing XML-encoded data. In SOAP messages, the name of the service request and the input parameters take the form of XML elements.

A SOAP message contains the following parts:

- ➢ The envelope—the top-level XML element
- ➢ The header—metadata about the SOAP message
- ➢ The body—the data that is being sent for the receiver to process.

Components of Web Services

SOAP Message Structure



```
<Soap: Envelope>
<Soap: Header>
---
---
</Soap: Header>
<Soap: Body>
---
---
</Soap: Body>
</Soap: Envelope>
```

Components of Web Services

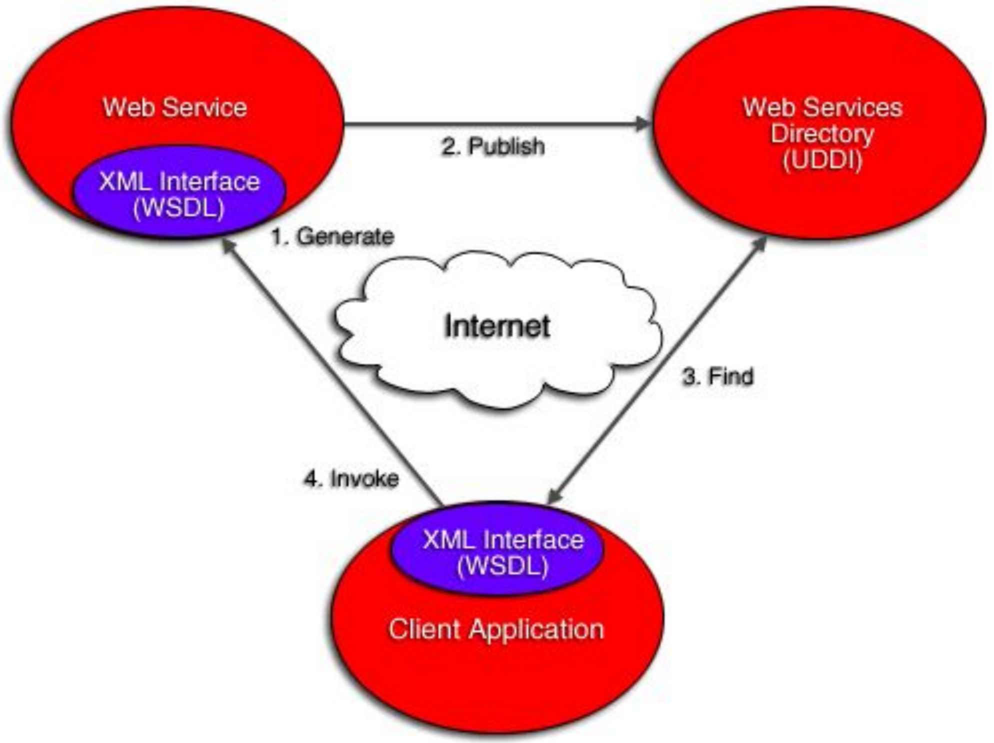
WSDL (Web Services Description Language):

WSDL allows service providers to specify what a web service can do, where it resides, and how to invoke it. WSDL describes the data types and structures for Web services, and tells how to map them into the messages that are exchanged.

UDDI (Universal Description, Discovery, and Interoperability):

UDDI provides a mechanism for clients to dynamically find other Web services. UDDI provides a repository for Web-services descriptions. An UDDI registry can be searched on various criteria to find all kinds of services offered by businesses

Components of Web Services



REST

REST, short for representational state transfer, is a type of software architecture that was designed to ensure interoperability between different Internet computer systems.

REST works by putting in place very strict constraints for the development of web services.

An API, or *application programming interface*, is a set of rules that define how applications or devices can connect to and communicate with each other.

All web services are APIs, but not all APIs are web services.

A REST API is an API that conforms to the design principles of the REST, or *representational state transfer* architectural style. For this reason, REST APIs are sometimes referred to RESTful APIs.

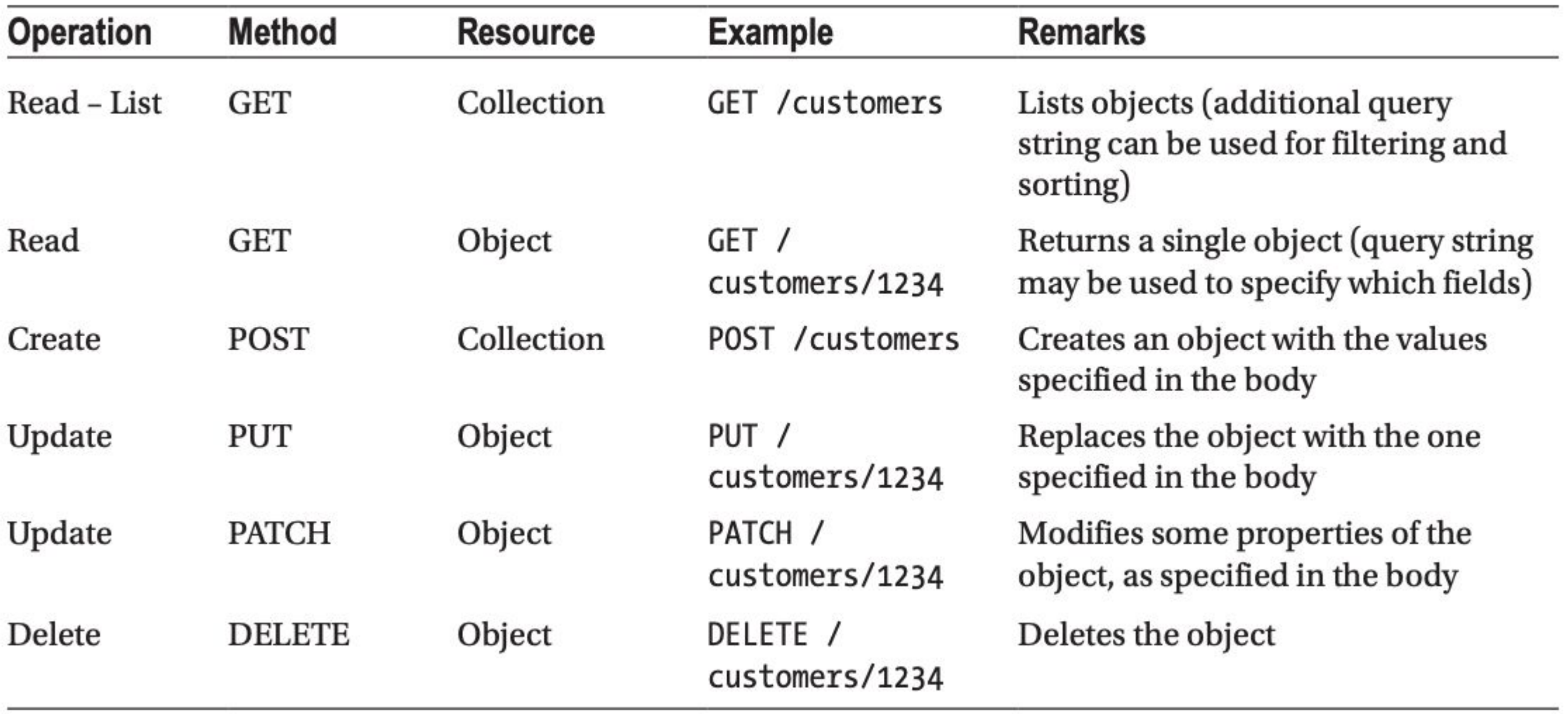
REST Design Principles

- **Client-server:** The client-server design pattern enforces the separation of concerns, which helps the client and the server components evolve independently.
- **Stateless:** As per the REST architecture, a RESTful Web Service should not keep a client state on the server. This restriction is called Statelessness.
- **Cacheable:** The cacheable constraint requires that a response should implicitly or explicitly label itself as cacheable or non-cacheable.

REST Design Principles

- ➢ **Uniform interface:** Resources are just concepts located by URIs. URIs tell client that there’s a concept somewhere. Client then asks for specific representation of the concept from the representations the server makes available.
- ➢ **Layered system:** The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior.
- ➢ **Code on demand (optional):** REST also allows client functionality to extend by downloading and executing code in the form of applets or scripts.

HTTP Methods as Actions



Operation	Method	Resource	Example	Remarks
Read – List	GET	Collection	`GET /customers`	Lists objects (additional query string can be used for filtering and sorting)
Read	GET	Object	`GET /customers/1234`	Returns a single object (query string may be used to specify which fields)
Create	POST	Collection	`POST /customers`	Creates an object with the values specified in the body
Update	PUT	Object	`PUT /customers/1234`	Replaces the object with the one specified in the body
Update	PATCH	Object	`PATCH /customers/1234`	Modifies some properties of the object, as specified in the body
Delete	DELETE	Object	`DELETE /customers/1234`	Deletes the object

SOAP vs REST

- ➢ SOAP is a protocol, whereas REST is an architectural style.
- ➢ SOAP requires more bandwidth and computing power than REST.
- ➢ SOAP provides better security than REST.
- ➢ SOAP supports only XML messages while REST supports plain text, XML, HTTP, JSON and many more message formats.
- ➢ SOAP provides lesser flexibility than REST.
- ➢ SOAP uses transfer protocols like HTTP, SMTP, UDP and many more while REST uses only HTTP.

Building RESTful APIs in MERN

ExpressJS is one of the most used packages by developers to build web APIs.

Express is a really cool Node framework that’s designed to help JavaScript developers create servers really quickly.

It provides mechanisms to:

- ➢ Write handlers for requests with different HTTP verbs at different URL paths (routes).
- ➢ Integrate with "view" rendering engines in order to generate responses by inserting data into templates.
- ➢ Set common web application settings like the port to use for connecting, and the location of templates that are used for rendering the response.
- ➢ Add additional request processing "middleware" at any point within the request handling pipeline.

Installing Express

Assuming you’ve already installed Node.js, to install the express.js package in your system globally, you can use the below command:

```
npm install -g express
```

Or, if you want to install it locally into your project folder, you need to execute the below command in your project environment:

```
npm install express --save
```

A Simple ExpressJS Server

```
const express = require('express');
const app = express();
app.listen(3000, () => {
   console.log('Server running on
localhost:3000');
 });
```

- An Express application is a web server that listens on a specific IP address and port. Multiple applications can be created, which listen on different ports, but we won't do that, as we need just a single server which is created by instantiating the one and only application by calling the express() function.
- The listen() method of the application starts the server and waits eternally for requests. It takes in a port number as the first argument.
- The listen() method also takes another argument, an optional callback that can be called when the server has been successfully started.

Routing

Routing refers to the process of determining a specific behavior of an application.

It is used for defining how an application should respond to a client request to a particular route, path or URI along with a particular HTTP Request method.

At the heart of Express is a router, which essentially takes a client request, matches it against any routes that are present, and executes the handler function that is associated with that route.

The handler function is expected to generate the appropriate response.

Routing

Below is the structure of Routing in Express:

```
app.METHOD(PATH, HANDLER)
```

Where:

- app is just an instance of Express.js. You can use any variable of your choice
- METHOD is an HTTP request method such as get, set, put, delete, etc.
- PATH is the route to the server for a specific webpage
- HANDLER is the callback function that is executed when the matching route is found.

Routing

Method	Description
1. GET	The HTTP GET method helps in requesting for the representation of a specific resource by the client. The requests having GET just retrieves data and without causing any effect.
2. POST	The HTTP POST method helps in requesting the server to accept the data that is enclosed within the request as a new object of the resource as identified by the URI.
3. PUT	The HTTP PUT method helps in requesting the server to accept the data that is enclosed within the request as an alteration to the existing object which is identified by the provided URI.
4. DELETE	The HTTP DELETE method helps in requesting the server to delete a specific resource from the destination.

Routing

```
var express = require('express');
var app = express();


app.get('/', function (req, res) {
 res.send('Home Page');
});
app.get('/about', function (req, res) {
   res.send('About Page');
 });
app.listen(3000);
```

The handler is passed in a request object and a response object. The request object can be inspected to get the various details of the request, and the response object's methods can be used to send the response to the client.

URL http://localhost:3000/ match

URL http://localhost:3000/about match

Routing

```
var express = require('express');
var app = express();

app.get('/getabout', function (req, res) {
   res.send("GET Request Received at /getabout");
 });
app.post('/postfeedback', function (req, res) {
   console.log("POST Request Received at /postfeedback");
   res.send('Feedback Page');
})
app.delete('/deletepost', function (req, res) {
   res.send('DELETE Request Received at /deletepost');
});
app.put('/post', function (req, res) {
 res.send('PUT Request Received at /post')
})
app.listen(3000);
```

app.all() matches any of the http methods at the specified route!

URL Building

Now that we are able to provide static routes, let’s move on to dynamic routes. Using dynamic routes allows us to pass parameters and process based on them.

Route parameters are named segments in the path specification that match a part of the URL.

If a match occurs, the value in that part of the URL is supplied as a variable in the request object. For instance:

```
app.get('/:id', function(req, res){
   res.send('The id you specified is ' + req.params.id);
});
```

The URL /1234 will match the route specification, and so will /4567. In either case, the ID will be captured and supplied to the handler function as part of the request in req. params, with the name of the parameter as the key. Thus, req.params.id will have the value 1234 or 4567 for each of these URLs, respectively.

More on the Request Object

Any aspect of the request can be inspected using the request object's properties and methods. A few important and useful properties and methods are listed here:

- **req.params:** This is an object containing properties mapped to the named route parameters as you saw in the example that used :ID. The property's key will be the name of the route parameter (ID in this case) and the value will be the actual string sent as part of the HTTP request.

- **req.query:** This holds a parsed query string. It's an object with keys as the query string parameters and values as the query string values.

- **req.path:** This contains the path part of the URL, that is, everything up to any ? that starts the query string.

- **req.url, req.originalURL:** These properties contain the complete URL, including the query string.

- **req.body:** This contains the body of the request, valid for POST, PUT, and PATCH requests.

More on the Response Object

The response object is used to construct and send a response. Note that if no response is sent, the client is left waiting.

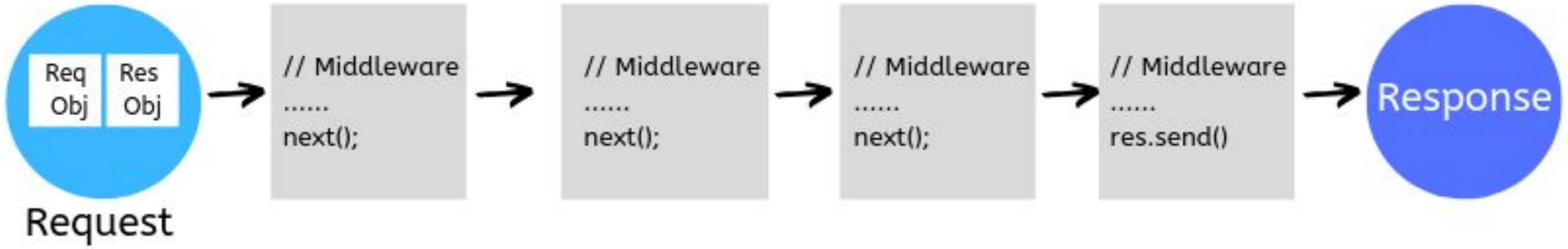
- ➢ **res.send(body):** You already saw the res.send() method briefly, which responded with a string. If the body is an object or an array, it is automatically converted to a JSON string with an appropriate content type.
- ➢ **res.status(code):** This sets the response status code. If not set, it is defaulted to 200 OK.
- ➢ **res.json(object):** This is the same as res.send(), except that this method forces conversion of the parameter passed into a JSON, whereas res.send() may treat some parameters like null differently.
- ➢ **res.sendFile(path):** This responds with the contents of the file at path. The content type of the response is guessed using the extension of the file.

Middleware

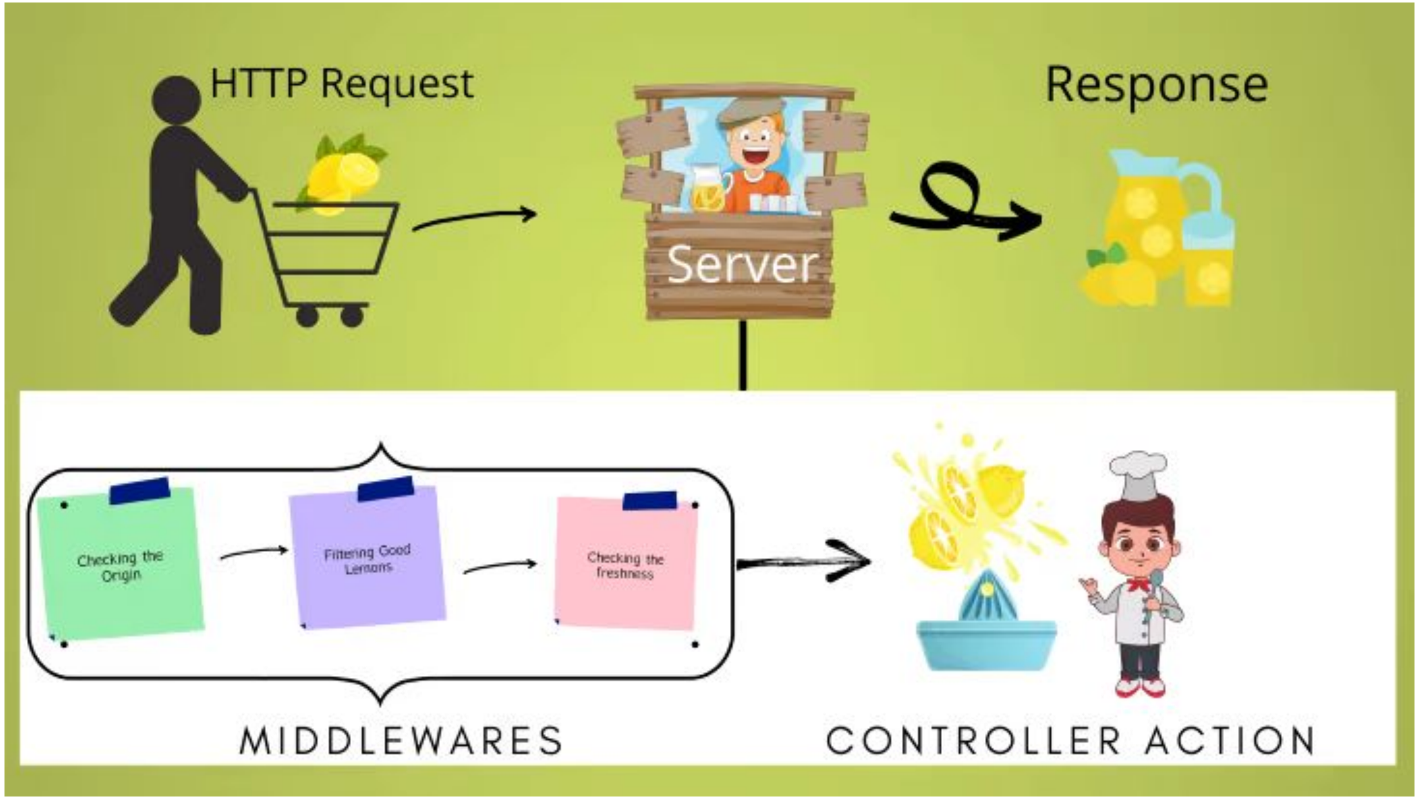
Express is a framework that does minimal work by itself; instead, it gets most of the job done by functions called middleware.

Middleware functions are those that have access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle. The next middleware function is commonly denoted by a variable named next.

The function can look at and modify the request and response objects, respond to requests, or decide to continue with middleware chain by calling the next middleware function



Middleware


HTTP Request
Server
Response
Checking the Origin
Filtering Good Lemons
Checking the freshness
MIDDLEWARES
CONTROLLER ACTION

Middleware

Middleware functions can perform the following tasks:

- ➢ Execute any code.
- ➢ Make changes to the request and the response objects.
- ➢ End the request-response cycle.
- ➢ Call the next middleware in the stack.

If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function. Otherwise, the request will be left hanging.

Middleware

Middleware can be at the application level (applies to all requests) or at a specific path level (applies to specific request path patterns).

The way to use a middleware at the application level is to simply supply the function to the application, like this:

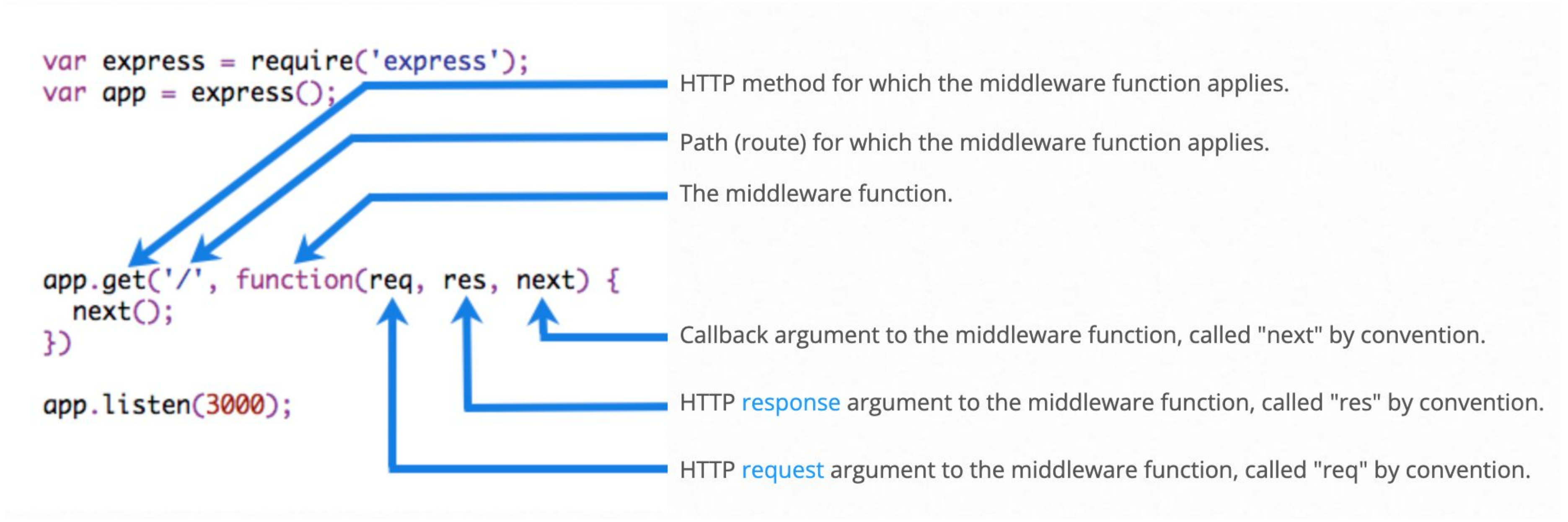
```
app.use(middlewareFunction);
```

In order to use the same middleware for only requests matching a certain URL path, say, /public, the app.use() method would have to be called with two arguments, the first one being the path, like this:

```
app.use('/public', middlewareFunction);
```

This would have mounted the static middleware on the path /public and all static files would have to be accessed with the prefix /public, for example, /public/index.html.

Anatomy of Middlewares



```
var express = require('express');
var app = express();

app.get('/', function(req, res, next) {
  next();
})

app.listen(3000);
```

HTTP method for which the middleware function applies.
Path (route) for which the middleware function applies.
The middleware function.
Callback argument to the middleware function, called "next" by convention.
HTTP response argument to the middleware function, called "res" by convention.
HTTP request argument to the middleware function, called "req" by convention.

Request Logging Middleware Example

```
const express = require('express');

const app = express();

app.use((req, res, next) => {

 console.log(req);

 next();

});


app.get('/', (req, res, next) => {

 res.send('Welcome Home');

});


app.listen(3000);
```

The middleware logs out the request object and then calls next(). The next middleware in the pipeline handles the get request to the root URL and sends back the text response.

When you go to http://localhost:3000 you should see the same thing in your browser window, but back in the console window you will see the output of the incoming request object!!

Request Time Logging Middleware Example

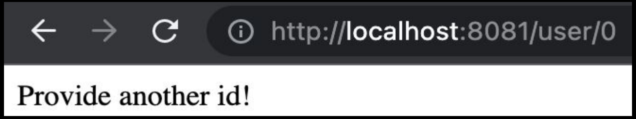
```
//logs request based on time
var express= require('express');
var app=express();

var requesttime= function(req,res,next){
   req.requestTime=Date.now();
   next(); //calls next middleware
}
app.use(requesttime);
app.get('/', function(req,res){
   var responseText='A request was made! <br> ';
   responseText+="Time of request : " + req.requestTime;
   res.send(responseText);
});
app.listen(8081);
```

Middleware function

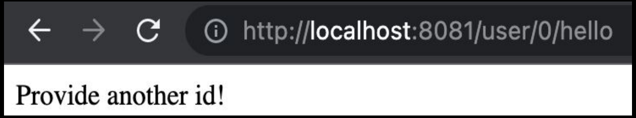
Middleware Working

```
var express= require('express');
var app=express();
app.use('/user/:id', function(req,res,next){
   if (req.params.id==0){
       res.send("Provide another id!");}
   else {
       next(); }
});
app.get("/",function(req,res){
   res.send("Welcome");
});
app.get("/user/:id",function(req,res){
       res.send("ID:  " + req.params.id);
});

app.get("/user/:id/hello",function(req,res){
   res.send("Hello ID:  " + req.params.id);
});
app.listen(8081, function(){console.log("Server is running")});
```

The middleware will perform its action of checking if the id is 0 for all subroutes of /user/:id .


http://localhost:8081/user/0

Provide another id!

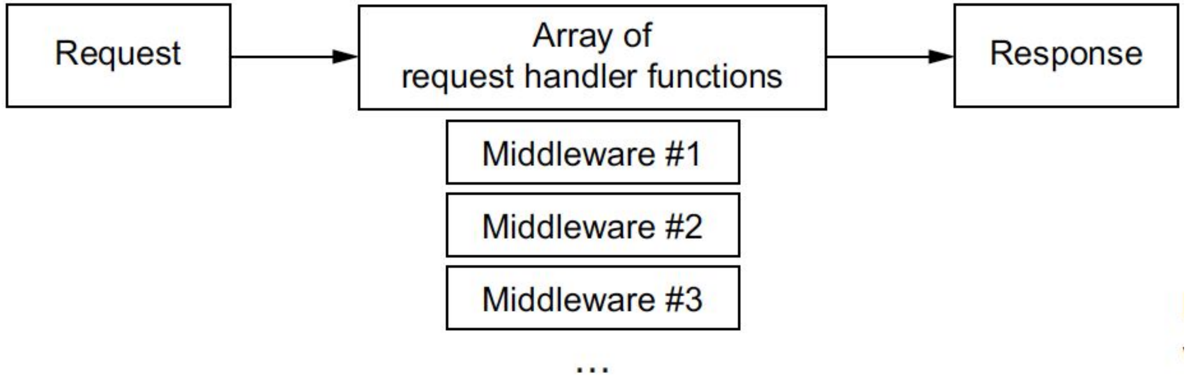

http://localhost:8081/user/0/hello

Provide another id!

Therefore, /user/0 and /user/0/hello will produce the same output as they have the prefix on which the middleware is applied.

Order of Middleware

One of the most important things about middleware in Express is the order in which they are written/included in your file; the order in which they are executed, given that the route matches also needs to be considered.

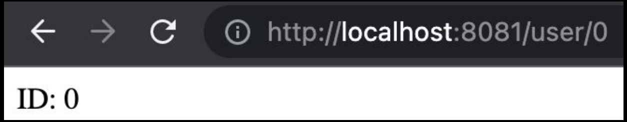


Order of Middleware

```
var express= require('express');
var app=express();
app.get("/user/:id",function(req,res){
   res.send("ID:  " + req.params.id);
});
app.use('/user/:id', function(req,res,next){
   if (req.params.id==0){
       res.send("Provide another id!");
   }
   else{
       next();
   }
});
app.get("/",function(req,res){
   res.send("Welcome");
});
app.get("/user/:id",function(req,res){
       res.send("ID:  " + req.params.id);
});
app.listen(8081, function(){ console.log("Server is running")});
```

Let's modify the previous example by placing the route before declaring the middleware.

When user/0 is provided as the route, the following is observed.


http://localhost:8081/user/0

ID: 0

In this case the middleware does not have access to the request as middlewares are executed in the order of placement.

Order of Middleware

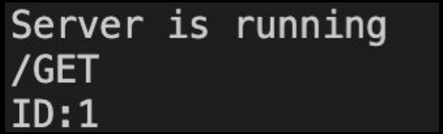
```
var express= require('express');
var app=express();

app.use(function(req,res,next){
   console.log("/" + req.method);
   next();
});
app.use('/user/:id', function(req,res,next){
   console.log("ID:" + req.params.id);
   next();
});
app.get("/",function(req,res){
   res.send("Welcome");
});
app.get("/user/:id",function(req,res){
       res.send("ID:  " + req.params.id);
});
app.listen(8080, function(){
   console.log("Server is running")
});
```

First middleware function that logs method

Second middleware function that logs id

The order of middleware loading is important: middleware functions that are loaded first are also executed first.

Console on http://localhost:8080/user/1



```
Server is running
/GET
ID:1
```

Order of Middleware

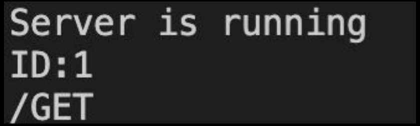
```
var express= require('express');
var app=express();

app.use('/user/:id', function(req,res,next){
   console.log("ID:" + req.params.id);
   next();
});
app.use(function(req,res,next){
   console.log("/" + req.method);
   next();
});
app.get("/",function(req,res){
   res.send("Welcome");
});
app.get("/user/:id",function(req,res){
       res.send("ID:  " + req.params.id);
});
app.listen(8080, function(){
   console.log("Server is running")
});
```

First middleware function that logs id

Second middleware function that logs method

Notice the change in the output on changing the order of the functions!

Console on http://localhost:8080/user/1



```
Server is running
ID:1
/GET
```

Third Party Middleware

Following are some of the most commonly used middleware:

- ➢ body-parser

This is used to parse the body of requests which have payloads attached to them. To mount body parser, we need to install it using npm install --save body-parser first.

- ➢ cookie-parser

It parses *Cookie* header and populate req.cookies with an object keyed by cookie names. To mount cookie parser, we need to install it using npm install --save cookie-parser first.

Error Handling

Error Handling refers to how Express catches and processes errors that occur both synchronously and asynchronously.

Express comes with a default error handler so you don't need to write your own to get started.

It is done using middleware but this middleware has special properties.

The error handling middleware is defined in the same way as other middleware functions, except that error-handling functions MUST have four arguments instead of three – err, req, res, next.

For error handling, we have the next(err) function. A call to this function skips all middleware and matches us to the next error handler for that route.

Error Handling

```
var express = require('express');
var app = express();

app.get('/', function(req, res){
  //Create an error and pass it to the next function
  var err = new Error("Something went wrong");
  next(err);
});
/*
* other route handlers and middleware here
* ....
*/

//An error handling middleware
app.use(function(err, req, res, next) {
  res.status(500);
  res.send("Oops, something went wrong.")
});
app.listen(3000);
```

This error handling middleware can be strategically placed after routes or contain conditions to detect error types and respond to the clients accordingly!

Handling Forms in Express

If you're building a web application, you're likely to encounter the need to build HTML forms

You can send form data in the following two ways:

GET: Data passed using the GET method will be visible in query parameters in browser URL.

POST: Data passed using the POST method will not visible in query parameters in browser URL.

By default, the enctype attribute **application/x-www-form-urlencoded** is used to submit standard form field.

Sending Data Using POST

```
<!-- form.html -->
<html>
   <head>
       <title>Form</title>
   </head>
   <body>
       <form method="POST"
action="/submit-form">
           <input type="text"
name="username" placeholder="Enter
Username" />
           <input type="submit" />
         </form>
   </body>
</html>
```

When the user presses the submit button, the browser will automatically make a POST request to the /submit-form URL on the same origin of the page.

The browser sends the data contained, encoded as application/x-www-form-urlencoded.

In this particular example, the form data contains the username input field value.

Sending Data Using POST

```
const express = require('express')
const app = express()
const path = require('path');
app.use(express.urlencoded({
 extended: true
}));
app.post('/submit-form', (req, res) => {
   const username = req.body.username
   res.send("<h1>Welcome " + username + "</h1>");
 });
app.get('/',function(req,res){
   res.sendFile(path.join(__dirname+'/form.html'));
 });
app.listen(8080);
```

form data will be sent in the POST request body. To extract it, you will need to use the express.urlencoded() middleware

data will be available on Request.body:

__dirname : It will resolve to your project folder.

Sending Data Using GET

```
<!--form_1.html-->
<html>
   <head>
       <title>Form</title>
   </head>
   <body>
       <form method="GET" action="/submit-form">
           <input type="text" name="username" placeholder="Enter Username" />
           <input type="submit" />
         </form>
   </body>
</html>
```

Sending Data Using GET

```
const express = require('express')
const app = express()
const path = require('path');
const url = require('url');
app.get('/submit-form', (req, res) => {
   const username = url.parse(req.url, true).query.username;
   res.send("<h1>Welcome " + username + "</h1>");
 });

app.get('/',function(req,res){
   res.sendFile(path.join(__dirname+'/form_1.html'));
 });
app.listen(8081);
```

data will be available in url in a get request

__dirname : It will resolve to your project folder.

Form-Data and File Upload

A large number of mobile apps and websites allow users to upload profile pictures and other files.

Therefore, handling files upload is a common requirement while building a REST API with Node.js & Express.

To upload files, install the express middleware for uploading files:

```
npm install express-fileupload
```

Uploading a File Through a Form

We often switch to multipart/form-data encoding type when we need to upload a file:

```
<!-- upload.html -->
<html>
 <body>
     <h1> Upload a file! </h1>
   <form
     action='http://localhost:8080/upload'
     method='post'
     encType="multipart/form-data">
       <input type="file" name="sampleFile" />
       <input type='submit'  />
   </form>
 </body>
</html>
```

where to send the form-data when a form is submitted

Form-Data and File Upload

When you upload a file, the file will be accessible from req.files.

Example:

- You're uploading a file called car.jpg
- Your input's name field is foo: <input name="foo" type="file" />
- In your express server request, you can access your uploaded file from req.files.foo:

This is the route that your form directs to (mentioned in the action attribute of your form)

```
app.post('/upload', function(req, res) {
   console.log(req.files.foo); // the uploaded file object
 });
```

Form-Data and File Upload

The req.files.foo object will contain the following:

- **`req.files.foo.name`**: "car.jpg"
- **`req.files.foo.mv`**: A function to move the file elsewhere on your server. Can take a callback or return a promise.
- **`req.files.foo.mimetype`**: The mimetype of your file
- **`req.files.foo.data`**: A buffer representation of your file, returns empty buffer in case useTempFiles option was set to true.
- **`req.files.foo.tempFilePath`**: A path to the temporary file in case useTempFiles option was set to true.
- **`req.files.foo.truncated`**: A boolean that represents if the file is over the size limit
- **`req.files.foo.size`**: Uploaded size in bytes
- **`req.files.foo.md5`**: MD5 checksum of the uploaded file

Form-Data and File Upload

```
//import required libraries
const express = require('express');
const fileUpload = require('express-fileupload');
const path= require('path');
const app = express();
//use middleware
app.use(fileUpload());
//display your form
app.get('/',function(req,res){
 res.sendFile(path.join(__dirname+'/upload.html'));
});
//continued
```

Form-Data and File Upload

```
//continued
app.post('/upload', function(req, res) {
    if (req.files){
     console.log(req.files); //view object structure
     var uploadedFile=req.files.sampleFile; // The name of the input field in your form
     var uploadedFile_name= uploadedFile.name;
     var uploadedFile_size= uploadedFile.size;
     //create a directory called uploads
     uploadedFile.mv('./uploads/'+ uploadedFile_name, function(err){
       if (err){
         console.log(err);
       }
       res.send(uploadedFile_name+ ' of size' + uploadedFile_size+ ' uploaded!');
     });
   }
});
app.listen(8080);
```

Templating in ExpressJS

Pug is a templating engine for Express.

Templating engines are used to remove the cluttering of our server code with HTML, concatenating strings wildly to existing HTML templates.

Pug is a very powerful templating engine which has a variety of features including filters, includes, inheritance, interpolation, etc.

To use Pug with Express, we need to install it:

```
npm install --save pug
```

Important Features of pug

Pug is **whitespace sensitive** which means that Pug uses indentation to work out which tags are nested inside each other.

Tags are nested according to their indentation.

We don’t need to close tags, as soon as Pug encounters the next tag on same or outer indentation level, it closes the tag for us.

Pug uses the same syntax as JavaScript(//) for creating comments. These comments are converted to the html comments(<!--comment-->).

Classes and IDs are expressed using a **.className** and **#IDname** notation

Using Pug

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Hello, World!</title>
  </head>
  <body>
    <h1>Hello, World!</h1>
    <div class="remark">
      <p>Pug rocks!!</p>
    </div>
  </body>
</html>
```

```pug
doctype html
html(lang='en')
 head
   title Hello, World!
 body
   h1 Hello, World!
   div.remark
     p Pug rocks!
```

THANK YOU

Prof. Spurthi N Anjan
Department of Computer Science and Engineering
spurthianjan@pes.edu

Sinduja Mullangi
Teaching Assistant
sinduja.mullangi@gmail.com